Appendix B

Edgeopt.cpp

Attorney Docket 3296.1

Inventor: Earl A. Hubbell


```
// Edge Optimizer
#include <cstring.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <time.h>
#include <math.h>

#define TAB '\t'
#define MXNAME 40
#define MXLINE 1000

#define TRUE 1
#define FALSE 0
#define MXSEQ 45
#define MXFEATURE 45
#define MXQUALIFIER 45

// Edge Optimizer Story
// 1) Strip off all Valid Blocks
// 2) Put Valid Blocks On to Minimize Edges

// Input: Cdl file, Ret file, Parameters
// Output: Twisted Cdl file and Ret file

char complement(char base)
{
    if (base=='A')
       return('T');
    if (base=='C')
       return('G');
    if (base=='G')
       return('C');
    if (base=='T')
       return('A');
    return(base);
}


class EntryClass{
// what CDL information is associated with everything
public:
       char sequence[MXSEQ];
       int destype;
       char feature[MXFEATURE];
```

```
        char qualifier[MXQUALIFIER];
        int expos;
        int     endpos;
        int pos;
        char pbase[MXFEATURE], tbase[MXFEATURE];
    int finishpos;
    int fixed;
    int variable;
        int unit, block;
    long atom;
    int repeat;
    int seqno;
    long layout;
        char locus[MXFEATURE];
    char accession[MXFEATURE];
    EntryClass(){Initialize();};
    Initialize();
    LineScan(char *);
    DumpLine(FILE *fp, int i, int j);
    DumpMut(FILE *fp);
};

EntryClass::Initialize()
{
        strcpy(sequence, "");
        destype = 0;
        strcpy(feature, "");
        strcpy(qualifier, "");
    expos = 0;
    pos = 0;
    strcpy(pbase,"!");
    strcpy(tbase,"!");
    unit = 0;
    block = 0;
    atom = 0;
}

EntryClass::LineScan(char *Line)
{
        int X,Y;
        static char    PROBE[MXLINE];
        int     DESTYPE;
        static char    FEATURE[MXLINE],
                       QUALIFIER[MXLINE];
        int     EXPOS;
        char    TBASE[MXLINE];
        int     ENDPOS,
                       POSITION;
        char  PBASE[MXLINE];
        int     FINISHPOS,
                       FIXED,
                       VARIABLE,
                       UNIT,
                       BLOCK,
                       REPEAT,
                       SEQNO,
                       LAYOUT;
    long ATOM;
        static char  ACCESSION[MXLINE],
                       LOCUS[MXLINE];

                       sscanf(Line, "%d %d %s %d %s %s %d %s %d %d %s %d %d %d %d %d %ld %d %d %d %s
%s",
                                                                &X,
                                                                &Y,
                                                                PROBE,
                                                                &DESTYPE,
                                                                FEATURE,
                                                                QUALIFIER,
                                                                &EXPOS,
```

```
                                                                TBASE,
                                                                &ENDPOS,
                                                                &POSITION,
                                                                PBASE,
                                                                &FINISHPOS,
                                                                &FIXED,
                                                                &VARIABLE,
                                                                &UNIT,
                                                                &BLOCK,
                                                                &ATOM,
                                                                &REPEAT,
                                                                &SEQNO,
                                                                &LAYOUT,
                                                                LOCUS,
                                                                ACCESSION);

                strcpy(sequence, PROBE);
                destype = DESTYPE;
                strcpy(feature, FEATURE);
                strcpy(qualifier, QUALIFIER);
                strcpy(locus, LOCUS);
                strcpy(accession, ACCESSION);
                expos = EXPOS;
                endpos = ENDPOS;
                        finishpos = FINISHPOS;
                fixed = FIXED;
                variable = VARIABLE;
                        repeat = REPEAT;
                        seqno = SEQNO;
                        layout = LAYOUT;

                        strcpy(tbase, TBASE);
                        strcpy(pbase, PBASE);
                        unit = UNIT;
                        block = BLOCK;
                        atom = ATOM;
                        pos = POSITION;
}

EntryClass::DumpLine(FILE *fp, int i, int j)
{
        fprintf(fp,
"%d\t%d\t%s\t%d\t%s\t%s\t%d\t%s\t%d\t%d\t%s\t%d\t%d\t%d\t%d\t%d\t%ld\t%d\t%d\t%d\t%s\t%s\n",
                                                                i,
                                                                j,
                                                                sequence,
                                                                destype,
                                                                feature,
                                                                qualifier,
                                                                expos,
                                                                tbase,
                                                                endpos,
                                                                pos,
                                                                pbase,
                                                                finishpos,
                                                                fixed,
                                                                variable,
                                                                unit,
                                                                block,
                                                                atom,
                                                                repeat,
                                                                seqno,
                                                                layout,
                                                                locus,
                                                                accession);
}

EntryClass::DumpMut(FILE *fp)
{
char tempc;

        if (destype==0)
```

```
{
    fprintf(fp, "-");
    return(TRUE);
}
if (abs(destype)<100)
{
   if (destype>0)
          fprintf(fp, "C");
    else
          fprintf(fp, "X");
    return(TRUE);
}
    if (strlen(pbase)>1)
{
    fprintf(fp, "I");
    return(TRUE);
}
if (pbase[0]=='!')
{
    fprintf(fp, "D");
    return(TRUE);
}
if (destype>0)
    {
       tempc = complement(pbase[0]);
    }
    else
       tempc = pbase[0];
    if (tempc==tbase[0])
    {
            fprintf(fp, "%c", tempc);
       return(TRUE);
    }
    fprintf(fp, " ");
    return(TRUE);
}


class SynthClass{
// how things are built
public:
    char *synthesis;
    int synlength;
       SynthClass(){synthesis=NULL; synlength = 0;};
    Allocate(int);
    DeAllocate();
    Diff(SynthClass &);
    SetBit(char, int);
    char GetBit(int);
    GetLast();
    GetFirst();
    ~SynthClass();
};

SynthClass::-SynthClass()
{
       DeAllocate();
}

SynthClass::Allocate(int Size)
{
       synthesis = new char [Size]; // just a bitfield, really
    if (synthesis==NULL)
    {
       printf("Blow up! In SynthClass::Allocate");
       return(FALSE);
    }
    for (int i=0; i<Size; i++)
       synthesis[i] = 0; // Nothing, null, no bits set!
    synlength = Size;
       return(TRUE);
```

```
}

SynthClass::DeAllocate()
{
        if (synthesis!=NULL)
                delete[] synthesis;
    synlength = 0;
    synthesis = NULL;
}

SynthClass::SetBit(char value, int Which)
{
        if (synthesis!=NULL && Which<synlength && Which>-1)
                synthesis[Which] = value;
}

char
SynthClass::GetBit(int Which)
{
        if (synthesis!=NULL && Which<synlength && Which>-1)
                return(synthesis[Which]);
    else
        return(0);
}

SynthClass::Diff(SynthClass &Source)
{
        int count = 0;
    if (!(synthesis!=NULL && Source.synthesis!=NULL && synlength==Source.synlength))
        return(0);
        for (int i=0; i<synlength; i++)
    {
                count += (synthesis[i]!=Source.synthesis[i]);
    }
    return(count);
}

SynthClass::GetLast()
{
        for (int i=synlength-1; i>0; i--)
        if (synthesis[i]>0)
                return(i);
}

SynthClass::GetFirst()
{
        for (int i=0; i<synlength; i++)
        if (synthesis[i]>0)
                return(i);
}

class LocalDataClass{
public:
        int relx, rely;  // relative positions within a block
    int validflag; // newly added - can I move this for optimization?
        EntryClass cdldata;
    SynthClass retdata;
    LocalDataClass(){relx=rely=0;validflag = TRUE;};
    ~LocalDataClass();
    SetRelative(int, int);
    PrintLongSeq(FILE *fp);
};

LocalDataClass::~LocalDataClass()
{
        retdata.DeAllocate();
}

LocalDataClass::SetRelative(int i, int j)
{
        relx = i;
```

```
    rely = j;
}

LocalDataClass::PrintLongSeq(FILE *fp)
{
        int j, i=3;

        for (j=0; j<retdata.synlength; j++)
    {
        if (retdata.GetBit(j))
        {
                fprintf(fp, "%c", cdldata.sequence[i]);
            i++;
        }
        else
                fprintf(fp, ".");
    }
}


class BlockClass{
public:
        LocalDataClass **DataStack;
    int DataStackSize;
    int WSize, HSize; // rectangular grid
    BlockClass(){DataStack=NULL; DataStackSize = 0; WSize = HSize = 0;};
    Allocate(int);
    DeAllocate();
    ~BlockClass();
};

BlockClass::Allocate(int Size)
{
        DataStack = new LocalDataClass *[Size];
    if (DataStack==NULL)
        return(FALSE);
    for (int i=0; i<Size; i++)
        DataStack[i]=NULL;
    DataStackSize = Size;
}

BlockClass::DeAllocate()
{
        if (DataStack==NULL)
        return(TRUE);
    for (int i=0; i<DataStackSize; i++)
        if (DataStack[i]!=NULL)
        {
                printf("Error!  Data Leakage from Block!\n");
                        delete DataStack[i];
            DataStack[i] = NULL;
        }
    delete[] DataStack;
    DataStack = NULL;
    DataStackSize = 0;
    WSize = HSize = 0;
}

BlockClass::~BlockClass()
{
        DeAllocate();
}

class BlockStackClass{
public:
        BlockClass **BlockStack;
    long BlockCurSize;
    long BlockStackSize;
    BlockStackClass(){BlockStack=NULL; BlockStackSize = 0;BlockCurSize = 0;};
    Allocate(long);
```

```
    DeAllocate();
    ~BlockStackClass(){DeAllocate();};
    BlockClass *PutBlockOnStack(BlockClass *);
    BlockClass *RemoveBlock(long);
    BlockClass *TemporaryBlockFromStack(long);
    Swap(long, long);
    Shuffle();
};

BlockStackClass::Allocate(long Size)
{
        BlockClass ** TmpStack;

        TmpStack = new BlockClass * [Size];
    if (TmpStack==NULL)
        return(FALSE);
    long i;

    for (i=0; i<Size; i++)
        TmpStack[i]=NULL;
    for (i=0; i<BlockCurSize && i<BlockStackSize && BlockStack!=NULL; i++)
    {
        TmpStack[i] = BlockStack[i];
        BlockStack[i] = NULL;
    }
    if (BlockStack!=NULL)
        delete[] BlockStack;
    BlockStack = TmpStack;
    TmpStack = NULL;
    BlockStackSize = Size;
    return(TRUE);
}

BlockClass *
BlockStackClass::PutBlockOnStack(BlockClass *TempBlock)
{
        if (BlockCurSize<BlockStackSize)
    {
    }
    else
    {
        if (!Allocate(BlockStackSize+1000))
        {
                printf("Can't increase block stack\n");
                return(TempBlock); // upgrade stack
        }
    }
    BlockStack[BlockCurSize] = TempBlock;
    BlockCurSize++;
    return(NULL); // remove pointer
}

BlockStackClass::Swap(long Source, long Sink)
{
        BlockClass *TempBlock;

    TempBlock = BlockStack[Sink];
    BlockStack[Sink] = BlockStack[Source];
    BlockStack[Source] = TempBlock;
    TempBlock = NULL;
    return(TRUE);
}

BlockClass *
BlockStackClass::RemoveBlock(long WhichBlock)
{
        BlockClass *TempBlock;

    if (WhichBlock>=BlockCurSize || WhichBlock<0)
        return(NULL);
```

```
        TempBlock = BlockStack[WhichBlock];
    BlockCurSize--;
    BlockStack[WhichBlock] = BlockStack[BlockCurSize];
    BlockStack[BlockCurSize]=NULL;
    return(TempBlock);
}

BlockClass *
BlockStackClass::TemporaryBlockFromStack(long WhichBlock)
{
        BlockClass *TempBlock;

    if (WhichBlock>=BlockCurSize || WhichBlock<0)
        return(NULL);

        TempBlock = BlockStack[WhichBlock];
    return(TempBlock);
}

BlockStackClass::DeAllocate()
{
        if (BlockStack==NULL)
        return(TRUE);
    long i;
    for (i=0; i<BlockStackSize; i++)
    {
        if (BlockStack[i]!=NULL)
        {
                printf("Data Leakage from BlockStack\n");
            delete BlockStack[i];
            BlockStack[i] = NULL;
        }
    }
    delete[] BlockStack;
    BlockStack = NULL;
    BlockStackSize = 0;
    BlockCurSize = 0;
}

BlockStackClass::Shuffle()
{
        long t;
    long i;

        // randomly rearrange the stack to prevent bias
    for (i=BlockCurSize-1; i>0; i--)
    {
        t = rand()%32000;
        t = t*32000+(rand()%32000);
        t = t %(i+1);
                Swap(i,t);
    }
}

class ChipArrayClass{
public:
        BlockStackClass ValidBlockStack;
    LocalDataClass ***DataGrid;
    int Xdim;
    int Ydim;
    int SynthSteps;

    char retname[MXNAME];
    long NumOnes; // useful statistic

    int GlobalHeight;
    int MaxAllowed;
    int Radius;

    float LeakageHalfLife;
    int ScanRadius;
```

```
int weightflag; // type of weights to use

// constraints


ChipArrayClass();
~ChipArrayClass();
Allocate(int, int);
DeAllocate();
ReadCdl(char *, int);
DumpCdl(char *);
ReadRet(char *, int);
DumpRet(char *);

StripAreaToBlock(BlockClass *, int,int,int,int);
CheckBlockFitToArea(BlockClass *, int, int);
   PutBlockInArea(BlockClass *, int, int);

ValidMove(int, int);
ValidLocation(int, int);
Valid(int,int);
ValidBlock(int,int,int,int);
ValidTile(int,int,int,int);
ValidBlank(int,int,int,int);

CountDiff(LocalDataClass *,int,int);
double CountEdges(BlockClass *, int, int);
double CountWeightedEdges(BlockClass *, int, int);
double CountEdgesFromStack(long, int, int);
FindNextDiagonalSlot(int &, int &);
FindNextHorizontalSlot(int &, int &);
DiagonalReplacement(long);
HorizontalReplacement(long);
StripAllValidBlocks(int);
PlaceBlockFromStack(long, int,int);
SearchLocationWithStats(int,int,long, long &, double &, double &, double &);

CountUnitInArea(long, int, int,int,int);
ProximityCheckBlock(BlockClass *, int,int,int,int);
ProximityCheckFromStack(long, int,int,int,int);

StripBadProximityValues(int);
PickRandomValidBlock(int &, int &, int, int);

ReadInstructionFile(char *);
InterpretInstructionLine(char *);
GenerateMutFile(char *);
SetUnits(long, long, int);
SetArea(int, int, int, int, int);
SetAntiArea(int,int,int,int,int);
SetDestype(int, int);
Shuffle();
StripValidBlock(int,int,int);
   StripRandomBlocks(long, int);
   DoubleReplacement(long);
   GenerateDiffFile(char *);

   FindNextAggregateSlot(int &, int &);
   AggregateReplacement(long);
};

ChipArrayClass::Shuffle()
{
      ValidBlockStack.Shuffle();
}

ChipArrayClass::SetArea(int X, int Y, int tX, int tY, int value)
{
      int i,j;
      for (i=X; i<=tX; i++)
   {
```

```
        for (j=Y; j<=tY; j++)
        {
                if (Valid(i,j))
                        DataGrid[i][j]->validflag = value;
        }
    }
}

ChipArrayClass::SetAntiArea(int X, int Y, int tX, int tY, int value)
{
        // used to set all >but< a given physical area
    int i,j;
    for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
                if (Valid(i,j))
            {
                if (!(i>=X && i<=tX && j>=Y && j<=tY))
                        DataGrid[i][j]->validflag = value;
            }
        }
    }
}

ChipArrayClass::SetDestype(int destype, int value)
{
        int i,j;

        for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
                if (Valid(i,j))
            {
                if (destype==DataGrid[i][j]->cdldata.destype)
                {
                        DataGrid[i][j]->validflag = value; // can't move this one
                }
            }
        }
    }

}

ChipArrayClass::ValidMove(int X, int Y)
{
        if (DataGrid[X][Y]->validflag)
        return(TRUE);
    return(FALSE);
}

ChipArrayClass::ValidLocation(int X, int Y)
{
        if (X<0 || Y<0 || X>=Xdim || Y>=Ydim)
        return(FALSE);
        // nothing here yet
    return(TRUE);
}

ChipArrayClass::Valid(int X, int Y)
{
    if (!ValidLocation(X,Y))
        return(FALSE); // not a location we're allowed to move
    if (DataGrid[X][Y]==NULL)
        return(FALSE); // doesn't even exist!
    return(TRUE);
}

ChipArrayClass::SetUnits(long Start, long Finish, int value)
{
```

```
        int i,j;

        for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
            if (Valid(i,j))
            {
            if (Start<=DataGrid[i][j]->cdldata.unit && Finish>=DataGrid[i][j]->cdldata.unit)
             {
                 DataGrid[i][j]->validflag = value; // can't move this one
             }
            }
        }
    }
}


ChipArrayClass::ValidTile(int X, int Y, int Width, int Height)
{
        long U, A;
    int tx, ty,i,j;
    int xl, yl;
    xl = X+Width;
    yl = Y+Height;

        U = DataGrid[X][Y]->cdldata.unit;
    A = DataGrid[X][Y]->cdldata.atom;
//    if (Width==Height && Height==1)  // if we're moving control probes around!
//            return(TRUE);
        for (i=-1; i<Width+1; i++)
    {
        for (j=-1; j<Height+1; j++)
        {
                    tx = i+X;
            ty = j+Y;
            if (tx>=X && ty>=Y && tx<xl && ty<yl)
            {
               if (!Valid(tx,ty) || !ValidMove(tx,ty))   // if not valid we're unhappy
                    return(FALSE);
                if (U!=DataGrid[tx][ty]->cdldata.unit || A!=DataGrid[tx][ty]->cdldata.atom) // if
not same, we're unhappy
                    return(FALSE);
            }
            else
            {
               if (Valid(tx,ty))
                {
                    if (U==DataGrid[tx][ty]->cdldata.unit && A==DataGrid[tx][ty]->cdldata.atom)
// if outside same block unhappy
                    return(FALSE);
                }
            }
        }
    }
    return(TRUE);
}

ChipArrayClass::ValidBlank(int X, int Y, int Width, int Height)
{
        int i,j, tx,ty;

        // valid set of blanks
    for (i=0; i<Width; i++)
        for (j=0; j<Height; j++)
        {
               tx = i+X;
            ty = j+Y;
            if (!Valid(tx,ty) || !ValidMove(tx,ty))
               return(FALSE);
            if (DataGrid[tx][ty]->cdldata.destype!=0)
```

```
                return(FALSE);
            }
        return(TRUE);
}

ChipArrayClass::ValidBlock(int X, int Y, int Width, int Height)
{

        if (!Valid(X,Y))
        return(FALSE);
    if (!ValidMove(X,Y))
       return(FALSE);
    if (ValidTile(X,Y,Width, Height))
        return(TRUE);
    if (ValidBlank(X,Y, Width, Height)) // allow blank blocks to be moved, if validflag set for
destype 0
        return(TRUE);

    return(FALSE);
}

ChipArrayClass::PickRandomValidBlock(int &X, int &Y, int Width, int Height)
{
        long counter = 0;
    long Limit = 10000;
        X = rand()%Xdim;
    Y = rand()%Ydim;
        while(!ValidBlock(X,Y, Width, Height) && counter<Limit)
    {
        X = rand()%Xdim;
        Y = rand()%Ydim;
        counter++;
    }
    if (counter==Limit)
       return(FALSE); // can't find one in reasonable time!
    return(TRUE);
}


ChipArrayClass::CountDiff(LocalDataClass *TestData, int X, int Y)
{
        if (!Valid(X,Y))
        return(0);  // doesn't exist or is off chip, so no problems!
    return(TestData->retdata.Diff(DataGrid[X][Y]->retdata));
}

double
ChipArrayClass::CountEdges(BlockClass *TempBlock, int X, int Y)
{
        int i,tx,ty;
    int count = 0;
       // count the edges, if this block is in this location
       // note that "interior" edges of blocks are >not< counted
       for (i=0; i<TempBlock->DataStackSize; i++)
       {
        if (TempBlock->DataStack[i]!=NULL)
        {
                tx = X+TempBlock->DataStack[i]->relx;
                ty = Y+TempBlock->DataStack[i]->rely;
            count +=CountDiff(TempBlock->DataStack[i], tx+1, ty);
            count +=CountDiff(TempBlock->DataStack[i], tx-1, ty);
            count +=CountDiff(TempBlock->DataStack[i], tx, ty-1);
            count +=CountDiff(TempBlock->DataStack[i], tx, ty+1);
        }
       }
    return(count);
}

double
ChipArrayClass::CountWeightedEdges(BlockClass *TempBlock, int X, int Y)
{
```

```
    int i,tx,ty;
int rangex,rangey;
double count = 0;
double lcount;
double distance;
    // count the edges, if this block is in this location
// note that "interior" edges of blocks are >not< counted
    for (i=0; i<TempBlock->DataStackSize; i++)
{
    if (TempBlock->DataStack[i]!=NULL)
    {
            for (rangex = -1*ScanRadius; rangex<=ScanRadius; rangex++)
        {
            for (rangey=-1*ScanRadius; rangey<=ScanRadius; rangey++)
             {
                if (rangex!=0 || rangey!=0)
                {
                        distance = (rangex*rangex)+(rangey*rangey);
                    distance = sqrt(distance);
                            tx = X+TempBlock->DataStack[i]->relx+rangex;
                        ty = Y+TempBlock->DataStack[i]->rely+rangey;
                         lcount =CountDiff(TempBlock->DataStack[i], tx, ty);
                        lcount*=pow(.5,LeakageHalfLife*distance);
                        count +=lcount;
                }
            }
        }
    }
    return(count);
}


ChipArrayClass::CountUnitInArea(long Unit, int X, int Y, int Width, int Height)
{
        int i,j;
    int tx, ty;
    int count =0;
    long tatom=-100; // unlikely in any real unit

        for (i=0; i<Width; i++)
    {
        tatom = -100; // start over with each vertical stripe - assumes vertical "atoms"
        for (j=0; j<Height; j++)
         {
                tx = X+i;
            ty= Y+j;
            if (Valid(tx,ty))
            {
                if (DataGrid[tx][ty]->cdldata.unit==Unit)
                 {
                        // works because we're scanning vertically
                        if (tatom!=DataGrid[tx][ty]->cdldata.atom) // only count a given unit/atom
once
                            count++;
                    tatom = DataGrid[tx][ty]->cdldata.atom;
                }
            }
        }
    }
    return(count);
}

ChipArrayClass::ProximityCheckBlock(BlockClass *TempBlock, int X, int Y, int Width, int Height)
{
        return(CountUnitInArea(TempBlock->DataStack[0]->cdldata.unit,X,Y,Width,Height));
}

ChipArrayClass::ProximityCheckFromStack(long Which, int X, int Y, int Width, int Height)
{
        BlockClass *TempBlock;
```

```
    int count;

    TempBlock = ValidBlockStack.TemporaryBlockFromStack(Which);
    if (TempBlock==NULL)
        return(16000); // big error
    count = ProximityCheckBlock(TempBlock, X, Y,Width, Height);
    TempBlock = NULL;
    return(count);
}

double
ChipArrayClass::CountEdgesFromStack(long Which, int X, int Y)
{
        BlockClass *TempBlock;
    double count;

    TempBlock = ValidBlockStack.TemporaryBlockFromStack(Which);
    if (TempBlock==NULL)
        return(16000); // big error
    if (!weightflag)
            count = CountEdges(TempBlock, X, Y);
    else
        count = CountWeightedEdges(TempBlock, X,Y);
    TempBlock = NULL;
    return(count);
}

ChipArrayClass::FindNextDiagonalSlot(int &X, int &Y)
{
        // look for a NULL entry
    while (Y<Ydim && DataGrid[X][Y]!=NULL)
    {
        X-=1;
        Y+=1;
        if (X<0 || Y>=Ydim)
        {
                X = Y+X+1;
            Y = 0;
        }
        if (X>=Xdim)
        {
                Y = X-(Xdim-1);
            X = Xdim-1;
        }
    }
    if (Y==Ydim) // ran out of room!
        return(FALSE);
    else
        return(TRUE);
}

ChipArrayClass::FindNextHorizontalSlot(int &X, int &Y)
{
        // look for a NULL entry
    while (Y<Ydim && DataGrid[X][Y]!=NULL)
    {
        X +=1;
        if (X>=Xdim)
        {
                X = 0;
            Y++;
        }
    }
    if (Y==Ydim) // ran out of room!
        return(FALSE);
    else
        return(TRUE);
}

ChipArrayClass::FindNextAggregateSlot(int &X, int &Y)
{
```

```
    int total;
    total = X;
if (Y>total)
    total = Y;
while (Y<Ydim && DataGrid[X][Y]!=NULL)
{
    // look for slots
    if (X>Y)
            Y++; // move vertically
    else
             if (X<=Y)
                    X--;  // basic zero moves
    if (X<0)
    {
            X=Y+1; // add one to total
        Y=0;
    }
    if (X>=Xdim)
    {
            Y = X;
        X=Xdim-1;
    }
    if (Y>=Ydim)
       return(FALSE);
    else
          return(TRUE);
}

ChipArrayClass::PlaceBlockFromStack(long Which, int X, int Y)
{
        BlockClass *TempBlock;

              TempBlock = ValidBlockStack.RemoveBlock(Which);
            if (TempBlock!=NULL)
            {
                              if (PutBlockInArea(TempBlock, X,Y))
                        TempBlock = NULL; // keep wacky pointers from drifting
                else
                {
                      printf("Failure to fit block in area: %d %d\n", X, Y);
                    TempBlock = ValidBlockStack.PutBlockOnStack(TempBlock); // throw back on stack
                }
            }
            else
               printf("Failure to get from stack! %d %d\n", X,Y);
}

ChipArrayClass::SearchLocationWithStats(int X, int Y, long searchlimit, long &Best, double &bestc,
double &avg, double &worstc)
{
        long search;
        long count = 0;
        double c;

            Best = ValidBlockStack.BlockCurSize-1; // which one
            bestc = CountEdgesFromStack(ValidBlockStack.BlockCurSize-1,X,Y);
                  avg = bestc;
            worstc = bestc;
            count = 1;
            for (search=1; search<searchlimit && search<ValidBlockStack.BlockCurSize; search++)
            {
                          c = CountEdgesFromStack(ValidBlockStack.BlockCurSize-1-search,X,Y); // what
if we put here?
                avg +=c;
                if (c<bestc)
                {
                   bestc = c;
                    Best = ValidBlockStack.BlockCurSize-1-search;
                }
                if (c>worstc)
```

```
                {
                    worstc=c;
                }
                count++;
            }
            avg /=count; // number actually searched
            return(TRUE);
}


ChipArrayClass::StripBadProximityValues(int H)
{
        int i,j;
    long U;
    int c;
    BlockClass *TempBlock;

    GlobalHeight = H;

        for (i=0; i<Xdim; i++)
    {
                for (j=0; j<Ydim; j++)
            {
                if (ValidBlock(i,j,1, H))  // note that blanks could mess this up badly!
                {
                    U = DataGrid[i][j]->cdldata.unit;
                        c = CountUnitInArea(U,i-Radius, j-Radius, 2*Radius+1, 2*Radius+1);
                    if (c>MaxAllowed)
                        StripValidBlock(i,j,H);
                }
            }
    }
    // now we've got all our trouble removed from the chip
    printf("Bad Proximity values: %d %d %d %ld\n", H, Radius, MaxAllowed,
ValidBlockStack.BlockCurSize);
    return(TRUE);
}



ChipArrayClass::DoubleReplacement(long searchlimit)
{
        // idea is to "dilute" any bad values
    // this only works if the chip is sufficiently large
    // and there are sufficiently few bad items
        StripBadProximityValues(GlobalHeight);
        while (ValidBlockStack.BlockCurSize>0)
    {
        StripRandomBlocks(ValidBlockStack.BlockCurSize+100, GlobalHeight); // get some good random
locations freed up
        Shuffle(); // rearrange life
        DiagonalReplacement(searchlimit); // put 'em back, - if too much search, goes back exactly
to bad spots
                StripBadProximityValues(GlobalHeight); // find out if we've got them all
    }
}


ChipArrayClass::DiagonalReplacement(long searchlimit)
{
        // replaces blocks from stack onto the chip
    int X, Y;
    long Best;
    double EdgesAdded = 0.1;
    double TotalAdded = 0.1;
    double AvgEdges =0.1;
    double WorstEdges=0.1;
    double bestc;
    double avg;
```

```
    double worstc;
    long Report = 1000000;

    Report /=searchlimit;
    if (Report>1000)
       Report=1000;

    X=Y=0;
    while (FindNextDiagonalSlot(X,Y))
    {
        // found a location where a block was removed
        SearchLocationWithStats(X,Y,searchlimit, Best, bestc, avg, worstc);
        // found the best thing to put there

        // and so put it there!
        PlaceBlockFromStack(Best,X,Y);

        EdgesAdded += bestc; AvgEdges += avg; WorstEdges += worstc; TotalAdded ++;

        if (ValidBlockStack.BlockCurSize%Report==0)
              printf("At: %d %d %ld %lf %lf %lf\r", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    }
    printf("\nAt: %d %d %ld %lf %lf %lf\n", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    return(TRUE);
}

ChipArrayClass::AggregateReplacement(long searchlimit)
{
        // replaces blocks from stack onto the chip
    int X, Y;
    long Best;
    double EdgesAdded = 0.1;
    double TotalAdded = 0.1;
    double AvgEdges =0.1;
    double WorstEdges=0.1;
    double bestc;
    double avg;
    double worstc;
    long Report = 1000000;

    Report /=searchlimit;
    if (Report>1000)
       Report=1000;

    X=Y=0;
    while (FindNextAggregateSlot(X,Y) && (ValidBlockStack.BlockCurSize>0))
    {
        // found a location where a block was removed
        SearchLocationWithStats(X,Y,searchlimit, Best, bestc, avg, worstc);
        // found the best thing to put there

        // and so put it there!
        PlaceBlockFromStack(Best,X,Y);

        EdgesAdded += bestc; AvgEdges += avg; WorstEdges += worstc; TotalAdded ++;

        if (ValidBlockStack.BlockCurSize%Report==0)
              printf("At: %d %d %ld %lf %lf %lf\r", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    }
    printf("\nAt: %d %d %ld %lf %lf %lf\n", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    return(TRUE);
}


ChipArrayClass::HorizontalReplacement(long searchlimit)
{
```

```
    // replaces blocks from stack onto the chip
int X, Y;
long Best;
double EdgesAdded = 0.1;
double TotalAdded = 0.1;
double AvgEdges =0.1;
double WorstEdges=0.1;
double bestc;
double avg;
double worstc;
long Report = 1000000;

Report /=searchlimit;
if (Report>1000)
    Report=1000;

X=Y=0;
while (FindNextHorizontalSlot(X,Y))
{
    // found a location where a block was removed
    SearchLocationWithStats(X,Y,searchlimit, Best, bestc, avg, worstc);
    // found the best thing to put there
        // and so put it there!
        PlaceBlockFromStack(Best,X,Y);

        EdgesAdded += bestc; AvgEdges += avg; WorstEdges += worstc; TotalAdded++;

        if (ValidBlockStack.BlockCurSize%Report==0)
              printf("At: %d %d %ld %lf %lf %lf\r", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    }
    printf("\nAt: %d %d %ld %lf %lf %lf\n", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    return(TRUE);
}

ChipArrayClass::StripValidBlock(int X, int Y, int H)
{
       BlockClass *TempBlock;

    if (ValidBlock(X,Y, 1, H))
    {
        TempBlock = new BlockClass;
        StripAreaToBlock(TempBlock, X,Y,1,H); // take probe from chip
        TempBlock = ValidBlockStack.PutBlockOnStack(TempBlock);
        if (TempBlock!=NULL)
        {
              printf("Failure to put on stack! %d %d\n", X,Y);
        }
    }
}

ChipArrayClass::StripAllValidBlocks(int H)
{
       int i,j;
    BlockClass *TempBlock;

       for (i=0; i<Xdim; i++)
    {
       for (j=0; j<Ydim; j++)
        {
              StripValidBlock(i,j,H);
        }
        printf("Stripped: %ld\r", ValidBlockStack.BlockCurSize);
    }
}

ChipArrayClass::StripRandomBlocks(long Num, int H)
{
       long i;
```

```
    int X,Y;

        for (i=0; i<Num; i++)
    {
        if(PickRandomValidBlock(X,Y,1,H))
                StripValidBlock(X,Y,H);
    }
    return(TRUE);
}


ChipArrayClass::StripAreaToBlock(BlockClass *TempBlock, int X, int Y, int Width, int Height)
{
        if (X+Width>Xdim || Y+Height>Ydim || X<0 || Y<0)
        return(FALSE);
        // strip an area of the chip into a block
    TempBlock->Allocate(Width*Height);
    TempBlock->WSize = Width;
    TempBlock->HSize = Height;

    int counter = 0;
    int i,j;
    for (i=0; i<Width; i++)
    for (j=0; j<Height; j++)
    {
        TempBlock->DataStack[counter] = DataGrid[X+i][Y+j];
        DataGrid[X+i][Y+j] = NULL; // removed
        TempBlock->DataStack[counter]->SetRelative(i,j);
        counter++;
    }
}

ChipArrayClass::CheckBlockFitToArea(BlockClass *TempBlock, int X, int Y)
{
        int valid = TRUE;
    int i;
    int tx, ty;

    if (TempBlock==NULL)
        return(FALSE);

        for (i=0; i<TempBlock->DataStackSize && valid; i++)
    {
        if (TempBlock->DataStack[i]!=NULL)
        {
                tx = X+TempBlock->DataStack[i]->relx;
                ty = Y+TempBlock->DataStack[i]->rely;
            if (tx<Xdim && ty<Ydim && tx>-1 && ty>-1)
                                if (DataGrid[tx][ty]!=NULL)
                        valid = FALSE;
                else
                        valid = TRUE;
                else
                    valid = FALSE;
        }
    }
    return(valid);
}

ChipArrayClass::PutBlockInArea(BlockClass *TempBlock, int X, int Y)
{
        int i;
    int tx, ty;
        if (!CheckBlockFitToArea(TempBlock, X, Y))
        return(FALSE);
        for (i=0; i<TempBlock->DataStackSize; i++)
    {
        if (TempBlock->DataStack[i]!=NULL)
        {
                tx = X+TempBlock->DataStack[i]->relx;
```

```
                    ty = Y+TempBlock->DataStack[i]->rely;
                        DataGrid[tx][ty] = TempBlock->DataStack[i];
                TempBlock->DataStack[i]=NULL;
            }
        }
            TempBlock->DeAllocate(); // toast!
            return(TRUE);
}

ChipArrayClass::ChipArrayClass()
{
            DataGrid = NULL;
        Xdim = Ydim = SynthSteps = 0;
        Radius = 9;
            MaxAllowed = 4;
        GlobalHeight = 2;

        ScanRadius = 1;
        LeakageHalfLife = 1;
        weightflag = 0;
}

ChipArrayClass::~ChipArrayClass()
{
            DeAllocate();
}

ChipArrayClass::Allocate(int X, int Y)
{
            DataGrid = new LocalDataClass ** [X];
        if (DataGrid==NULL)
            return(FALSE);
        int i,j;

        for (i=0; i<X; i++)
        {
            DataGrid[i] = new LocalDataClass * [Y];
            if (DataGrid[i]==NULL)
                    return(FALSE);
            for (j=0; j<Y; j++)
            {
                    DataGrid[i][j] = new LocalDataClass; // featherweight objects
                if (DataGrid[i][j]==NULL)
                    return(FALSE);
            }
        }
        Xdim = X;
        Ydim = Y;
        return(TRUE);
}

ChipArrayClass::DeAllocate()
{
            if (DataGrid==NULL)
            return(TRUE);
            int i,j;
            for (i=0; i<Xdim; i++)
        {
            for (j=0; j<Ydim && DataGrid[i]!=NULL; j++)
            {
                    if (DataGrid[i][j]!=NULL)
                            delete DataGrid[i][j];
            }
            if (DataGrid[i]!=NULL)
                    delete[] DataGrid[i];
        }
        delete[] DataGrid;
        DataGrid = NULL;
        Xdim = Ydim = 0;
        SynthSteps = 0;
}
```

```
ChipArrayClass::ReadCdl(char *FileName, int realflag)
{
        FILE *ifp;
    int maxX, maxY, X, Y;
    char datastring[MXLINE];
    int flag=TRUE;

    if (realflag)
            flag = ReadCdl(FileName, FALSE);
    if (!flag)
       return(FALSE);
       ifp = fopen(FileName, "rt");
    if (NULL==ifp)
    {
       printf("Unable to open: %s\n", FileName);
        exit(1);
    }
    fgets(datastring, MXLINE, ifp);
    maxX = 0;
    maxY = 0;
    while (!feof(ifp) && !ferror(ifp))
    {
        fgets(datastring, MXLINE, ifp);
        if (feof(ifp) || ferror(ifp))
                break;
        sscanf(datastring, "%d %d", &X, &Y);
        if (X>maxX)
                maxX=X;
        if (Y>maxY)
                maxY=Y;
        if (realflag)
                DataGrid[X][Y]->cdldata.LineScan(datastring);
        if (X==0)
                printf("%d\r", Y);
    }
    fclose(ifp);
    if(!realflag)
    {
            maxX++;
            maxY++;
        flag = Allocate(maxX, maxY);
        return(flag);
    }
    return(TRUE);
}

ChipArrayClass::DumpCdl(char *FileName)
{
        FILE *fp;
    int i,j;
        fp = fopen(FileName, "wt");
    fprintf(fp,
"X\tY\tPROBE\tDESTYPE\tFEATURE\tQUALIFIER\tEXPOS\tTBASE\tENDPOS\tPOSITION\tPBASE\tFINISHPOS\tFIXED\t
VARIABLE\tUNIT\tBLOCK\tATOM\tREPEAT\tSEQNO\tLAYOUT\tACCESSION\tLOCUS\n");
        for (j=0; j<Ydim; j++)
    {
                for (i=0; i<Xdim; i++)
        {
                if (DataGrid[i][j]!=NULL)
                        DataGrid[i][j]->cdldata.DumpLine(fp, i,j);
            else
            {
               printf("Null value in grid %d %d\n", i,j);
            }
        }
        printf("OutCdl: %d\r", j);
    }
        fclose(fp);
```

```
}

ChipArrayClass::GenerateMutFile(char *FileName)
{
        FILE *fp;
    int i,j;

        fp = fopen(FileName, "wt");
        for (j=0; j<Ydim; j++)
    {
                for (i=0; i<Xdim; i++)
            {
                if (DataGrid[i][j]!=NULL)
            {
                                DataGrid[i][j]->cdldata.DumpMut(fp); // single descriptive character
            }
            else
            {
                fprintf(fp, "-");
                printf("Null value in grid %d %d\n", i,j);
            }
        }
            fprintf(fp, "\n");
            printf("MUT: %d\r", j);
    }
        fclose(fp);
}

ChipArrayClass::GenerateDiffFile(char *FileName)
{
        FILE *fp;
    int i,j;
    int tn,te,ts,tw;
    double n,e,s,w;
    double count;

    fp = fopen(FileName, "wt");
    for (j=0; j<Ydim; j++)
    {
        for (i=0; i<Xdim; i++)
        {
               if (Valid(i,j))
            {
               fprintf(fp, "X:%d\tY:%d\t", i,j);
                tn=ts=tw=te =0;
                if (Valid(i,j-1))
                      tn = DataGrid[i][j-1]->retdata.Diff(DataGrid[i][j]->retdata);
                if (Valid(i,j+1))
                      ts = DataGrid[i][j+1]->retdata.Diff(DataGrid[i][j]->retdata);
                if (Valid(i-1,j))
                      tw = DataGrid[i-1][j]->retdata.Diff(DataGrid[i][j]->retdata);
                if (Valid(i+1,j))
                      te = DataGrid[i+1][j]->retdata.Diff(DataGrid[i][j]->retdata);
                fprintf(fp, "N:%d\tE:%d\tS:%d\tW:%d\tT:%d\t",tn,te,ts,tw,tn+te+ts+tw);
                        fprintf(fp, "LAST:%d\t", DataGrid[i][j]->retdata.GetLast());
                fprintf(fp, "BREADTH:%d\t", DataGrid[i][j]->retdata.GetLast()-DataGrid[i][j]-
>retdata.GetFirst());
                DataGrid[i][j]->PrintLongSeq(fp);
                fprintf(fp, "\t%s\n", DataGrid[i][j]->cdldata.qualifier);

                count++;
                n+=tn;
                s+=ts;
                e+=te;
                w+=tw;
            }
            else
            {
               printf("Null value in grid %d %d\n", i,j);
            }
        }
```

```
        printf("Diff: %d %lf %lf %lf %lf %lf\r", j, n/count, e/count,s/count, w/count,
(n+e+s+w)/(4*count));
    }
    //fprintf(fp,"Diff: %d %lf %lf %lf %lf %lf\n", j, n/count, e/count,s/count, w/count,
(n+e+s+w)/(4*count));
    fclose(fp);
}

ChipArrayClass::ReadRet(char *FileName, int realflag)
{
        FILE *fp;
    int i,j, k;
    char dataline[MXLINE];
    long total;

    if (realflag)
        ReadRet(FileName, 0);
    fp = fopen(FileName, "rt");
    if (fp==NULL)
    {
        printf("Unable to open: %s\n", FileName);
        exit(1);
    }

if (realflag)
{
    for (i=0; i<Xdim; i++)
    for (j=0; j<Ydim; j++)
    {
            if (DataGrid[i][j]!=NULL)
                        DataGrid[i][j]->retdata.Allocate(SynthSteps);  // allocate this data
        else
            printf("Death by lack of allocation\n");
    }
}


k=-1;
j=Ydim;
total = 0;

while (fgets(dataline, MXLINE, fp))
{
    if (dataline[0]=='r')
    {
            sscanf(dataline, "reticle: %s", &retname);  // set up reticle template name
        retname[strlen(retname)-2] = '\0';
                // initialize for reading next lines
        k++;
        printf("Reticle: %d\r", k);
        j=Ydim;
        continue;
    }
    if (strlen(dataline)<10 || dataline[0]==';')
        continue;
    if (dataline[0]=='0' || dataline[0]=='1')
    {
        j--;
        for (i=0; i<Xdim && j>-1; i++)
        {
        if (dataline[i]=='1')
        {
            if (realflag)
            {
                DataGrid[i][j]->retdata.SetBit(1,k);
            }
            total ++;
        }
        }
    }
}
```

```
    NumOnes = total;
    SynthSteps = k+1;

    fclose(fp);
}

ChipArrayClass::DumpRet(char *FileName)
{
      FILE *fp;
    int i,j,k;
    char dataline[MXLINE];

      fp = fopen(FileName, "wt");

    fprintf(fp, "; This file has been annealed to minimize edges\n");

    for (k=0; k<SynthSteps; k++)
    {
            fprintf(fp,"\n\n;base: X");
            fprintf(fp,"\nreticle: %s%02d", retname, (k+1));
            fprintf(fp,"\nR I 1 1 0 0 %d %d %d",Xdim, Ydim, 1);
      for (j=Ydim-1; j>-1; j--)
       {
          fprintf(fp, "\n");
             for (i=0; i<Xdim; i++)
          {
             if (DataGrid[i][j]==NULL)
              {
                    printf("Data leakage: %d %d\n", i,j);
              }
              else
                    if (DataGrid[i][j]->retdata.GetBit(k))
                          dataline[i] = '1';
                     else
                          dataline[i] = '0';
          }
          dataline[Xdim] = '\0';
          fprintf(fp, "%s", dataline);
       }
       fprintf(fp,";\n0;\n");
       printf("DUMPRET: %d\r", k);
    }
    fclose(fp);
}

ChipArrayClass::InterpretInstructionLine(char *Line)
{
      char TempStr[MXLINE];
    int height;
    long searchlimit;
    long start, finish;
    int tx,ty,x,y;
    int value;
    int destype;
    int radius, max;
    double dval;

    // read an instruction and do the appropriate thing
    if (Line[0]==';')
      return(TRUE); // comment
    sscanf(Line, "%s", TempStr); // pick off the initial piece
    if (!strcmp(TempStr, "READCDL:"))
    {
            sscanf(Line, "READCDL: %s", TempStr);
      ReadCdl(TempStr, TRUE);
      return(TRUE);
    }
    if (!strcmp(TempStr, "READRET:"))
    {
      sscanf(Line, "READRET: %s", TempStr);
      ReadRet(TempStr,1);
```

```
        return(TRUE);
    }
    if (!strcmp(TempStr, "DUMPCDL:"))
    {
        sscanf(Line, "DUMPCDL: %s", TempStr);
        DumpCdl(TempStr);
        return(TRUE);
    }
    if (!strcmp(TempStr, "DUMPRET:"))
    {
        sscanf(Line, "DUMPRET: %s", TempStr);
        DumpRet(TempStr);
        return(TRUE);
    }
    if (!strcmp(TempStr, "DUMPMUT:"))
    {
        sscanf(Line, "DUMPMUT: %s", TempStr);
        GenerateMutFile(TempStr);
        return(TRUE);
    }
    if (!strcmp(TempStr, "DUMPDIFF:"))
    {
        sscanf(Line, "DUMPDIFF: %s", TempStr);
        GenerateDiffFile(TempStr);
        return(TRUE);
    }
        if (!strcmp(TempStr, "STRIPBLOCKS:"))
    {
        sscanf(Line, "STRIPBLOCKS: %d", &height);
        GlobalHeight = height;
        StripAllValidBlocks(height);
        Shuffle();
        return(TRUE);
    }
    if (!strcmp(TempStr, "DIAGONALREPLACEMENT:"))
    {
        sscanf(Line, "DIAGONALREPLACEMENT: %ld", &searchlimit);
        DiagonalReplacement(searchlimit); // put all blocks back on chip
        return(TRUE);
    }
    if (!strcmp(TempStr, "HORIZONTALREPLACEMENT:"))
    {
        sscanf(Line, "HORIZONTALREPLACEMENT: %ld", &searchlimit);
        DiagonalReplacement(searchlimit); // put all blocks back on chip
        return(TRUE);
    }
    if (!strcmp(TempStr, "AGGREPLACEMENT:"))
    {
        sscanf(Line, "AGGREPLACEMENT: %ld", &searchlimit);
        AggregateReplacement(searchlimit); // put all blocks back on chip
        return(TRUE);
    }
    if (!strcmp(TempStr, "SETVALIDUNITS:"))
    {
        sscanf(Line, "SETVALIDUNITS: %ld %ld %d", &start, &finish, &value);
        SetUnits(start, finish, value);
        return(TRUE);
    }
    if (!strcmp(TempStr, "SETVALIDAREA:"))
    {
        sscanf(Line, "SETVALIDAREA: %d %d %d %d %d", &x,&y,&tx,&ty, &value);
        SetArea(x,y,tx,ty,value);
        return(TRUE);
    }
    if (!strcmp(TempStr, "SETVALIDANTIAREA:"))
    {
        sscanf(Line, "SETVALIDANTIAREA: %d %d %d %d %d", &x,&y,&tx,&ty, &value);
        SetAntiArea(x,y,tx,ty,value);
        return(TRUE);
    }
    if (!strcmp(TempStr, "SETVALIDDESTYPE:"))
```

```
    {
        sscanf(Line, "SETVALIDDESTYPE: %ld %d", &destype, &value);
        SetDestype(destype, value);
        return(TRUE);
    }
    if (!strcmp(TempStr, "STRIPBADPROXIMITY:"))
    {
        sscanf(Line, "STRIPBADPROXIMITY: %d %d %d", &height, &radius, &max);
        Radius = radius;
        MaxAllowed = max;
        StripBadProximityValues(height);
        return(TRUE);
    }
    if (!strcmp(TempStr, "SETPROXIMITY:"))
    {
        sscanf(Line, "SETPROXIMITY: %d %d", &radius, &max);
        Radius = radius;
        MaxAllowed = max;
        return(TRUE);
    }
    if (!strcmp(TempStr, "FIXBAD:"))
    {
        sscanf(Line, "FIXBAD: %ld", &searchlimit);
        DoubleReplacement(searchlimit);
        return(TRUE);
    }
    if (!strcmp(TempStr, "WEIGHT:"))
    {
        sscanf(Line, "WEIGHT: %d %lf", &radius, &dval);
        ScanRadius = radius;
        LeakageHalfLife = dval;
        weightflag = TRUE; // use weights
        return(TRUE);
    }
    if (!strcmp(TempStr, "NOWEIGHT:"))
    {
        weightflag = FALSE;
        return(TRUE);
    }
    return(FALSE);
}

ChipArrayClass::ReadInstructionFile(char *FileName)
{
        // read the file and be happy
    FILE *fp;
    char dataline[MXLINE];

    fp = fopen(FileName, "rt");
    if (fp==NULL)
        return(FALSE);
    while (fgets(dataline, MXLINE, fp))
    {
            InterpretInstructionLine(dataline);
    }
    fclose(fp);
}


TestTwo()
{
        ChipArrayClass Test;
    Test.ReadInstructionFile("test.opt");
}

Live(char *FileName)
{
        ChipArrayClass Test;
    Test.ReadInstructionFile(FileName);
}
```

```
main(int argc, char **argv)
{
    if (argc==2)
    {
        Live(argv[1]);
    }
    else
    {
        printf("File name required!");
    }
};
```